synchronization 3: monitors pt 2 / semaphores / rwlock

last time

barriers — everyone waits for everyone else

locking integrated with scheduler

- make non-runnable when waiting for lock
- unlock operation responsible for making runnable again

“condition variables” — queue of threads interface

- `pthread_cond_t`

monitors — locks + condition variables + shared data

- lock to protect shared state —- take turns
- condition variables to wait for something to change about state

life assignment

if you took CoA2 — additional requirement and for next assignment

sanitizer versions / sanitizer-test.sh script

- race condition (tsan) + memory error (asan)
- hopefully decreases number of submissions with race conditions?
- uploaded wrong version originally

monitor exercise: barrier

suppose we want to implement a one-use barrier

what goes in the blanks?

```
struct BarrierInfo {
    pthread_mutex_t lock;
    int total_threads;  // initially total # of threads
    int number_reached; // initially 0
    ___________________
};

void BarrierWait(BarrierInfo *b) {
    pthread_mutex_lock(&b->lock);
    ++b->number_reached;
    _____________________
    _____________________
    _____________________
    _____________________
    pthread_mutex_unlock(&b->lock);
}
```

monitor exercise: ConsumeTwo

suppose we want producer/consumer, but...

but change to ConsumeTwo() which returns a pair of values

- and don't want two calls to ConsumeTwo() to wait...
- with each getting one item

what should we change below?

```
pthread_mutex_t lock;
pthread_cond_t data_ready;
UnboundedQueue buffer;

Produce(item) {
  pthread_mutex_lock(&lock);
  buffer.enqueue(item);
  pthread_cond_signal(&data_ready);
  pthread_mutex_unlock(&lock);
}
```

```
Consume() {
  pthread_mutex_lock(&lock);
  while (buffer.empty()) {
    pthread_cond_wait(&data_ready, &lock)
  }
  item = buffer.dequeue();
  pthread_mutex_unlock(&lock);
  return item;
}
```

monitor exercise: solution (1)

(one of many possible solutions)

Assuming ConsumeTwo **replaces** Consume:

```
Produce() {
  pthread_mutex_lock(&lock);
  buffer.enqueue(item);
  if (buffer.size() > 1) { pthread_cond_signal(&data_ready); }
  pthread_mutex_unlock(&lock);
}
ConsumeTwo() {
    pthread_mutex_lock(&lock);
    while (buffer.size() < 2) { pthread_cond_wait(&data_ready, &lock); }
    item1 = buffer.dequeue(); item2 = buffer.dequeue();
    pthread_mutex_unlock(&lock);
    return Combine(item1, item2);
}
```

monitor exercise: solution (2)

(one of many possible solutions)

Assuming ConsumeTwo is **in addition to** Consume (using two CVs):

```
Produce() {
  pthread_mutex_lock(&lock);
  buffer.enqueue(item);
  pthread_cond_signal(&one_ready);
  if (buffer.size() > 1) { pthread_cond_signal(&two_ready); }
  pthread_mutex_unlock(&lock);
}
Consume() {
  pthread_mutex_lock(&lock);
  while (buffer.size() < 1) { pthread_cond_wait(&one_ready, &lock); }
  item = buffer.dequeue();
  pthread_mutex_unlock(&lock);
  return item;
}
ConsumeTwo() {
  pthread_mutex_lock(&lock);
  while (buffer.size() < 2) { pthread_cond_wait(&two_ready, &lock); }
  item1 = buffer.dequeue(); item2 = buffer.dequeue();
  pthread_mutex_unlock(&lock);
  return Combine(item1, item2);
}
```

monitor exercise: slower solution

(one of many possible solutions)

Assuming ConsumeTwo is **in addition to** Consume (using one CV):

```
Produce() {
  pthread_mutex_lock(&lock);
  buffer.enqueue(item);
  // broadcast and not signal, b/c we might wakeup only ConsumeTwo() otherwise
  pthread_cond_broadcast(&data_ready);
  pthread_mutex_unlock(&lock);
}
Consume() {
  pthread_mutex_lock(&lock);
  while (buffer.size() < 1) { pthread_cond_wait(&data_ready, &lock); }
  item = buffer.dequeue();
  pthread_mutex_unlock(&lock);
  return item;
}
ConsumeTwo() {
  pthread_mutex_lock(&lock);
  while (buffer.size() < 2) { pthread_cond_wait(&data_ready, &lock); }
  item1 = buffer.dequeue(); item2 = buffer.dequeue();
  pthread_mutex_unlock(&lock);
  return Combine(item1, item2);
}
```

monitor exercise: ordering

suppose we want producer/consumer, but...

but want to ensure first call to Consume() **always** returns first

(no matter what ordering cond_signal/cond_broadcast use)

```
pthread_mutex_t lock;
pthread_cond_t data_ready;
UnboundedQueue buffer;

Produce(item) {
  pthread_mutex_lock(&lock);
  buffer.enqueue(item);
  pthread_cond_signal(&data_ready);
  pthread_mutex_unlock(&lock);
}
```

```
Consume() {
  pthread_mutex_lock(&lock);
  while (buffer.empty()) {
    pthread_cond_wait(&data_ready, &lock)
  }
  item = buffer.dequeue();
  pthread_mutex_unlock(&lock);
  return item;
}
```

monitor ordering exercise: solution

(one of many possible solutions)

```
struct Waiter {
    pthread_cond_t cv;
    bool done;
}
Queue<Waiter*> waiters;

Produce() {
 pthread_mutex_lock(&lock);
 ...
 if (!waiters.empty()) {
   Waiter *waiter = waiters.dequeue();
   waiter->done = true;
   cond_signal(&waiter->cv);
 }
 ...
 pthread_mutex_unlock(&lock);
}
```

```
Consume() {
  pthread_mutex_lock(&lock);
  if (buffer.empty() || !waiters.empty
    Waiter waiter;
    cond_init(&waiter.cv);
    waiter.done = false;
    waiters.enqueue(&waiter);
    while (!waiter.done)
      cond_wait(&waiter.cv, &lock);
  }
  item = buffer.dequeue();
  pthread_mutex_unlock(&lock):
  return item;
}
```

waiter queue solution

not a very satisfying solution (in my opinion)

redoing what a condition variable does internally

- ...and using a bunch of condition variables to do it

if we had lower-level tools — should be more efficient?

generalizing locks: semaphores

semaphore has a non-negative integer **value** and two operations:

P() or **down** or **wait**:
wait for semaphore to become positive (> 0),
then decerement by 1

V() or **up** or **signal** or **post**:
increment semaphore by 1 (waking up thread if needed)

P, V from Dutch: *proberen* (test), *verhogen* (increment)

semaphores are kinda integers

semaphore like an integer, but...

cannot read/write directly

- down/up operaion only way to access (typically)
- exception: initialization

never negative — wait instead

- down operation wants to make negative? thread waits

reserving books

suppose tracking copies of library book...

```
Semaphore free_copies = Semaphore(3);
void ReserveBook() {
    // wait for copy to be free
    free_copies.down();
    ... // ... then take reserved copy
}

void ReturnBook() {
    ... // return reserved copy
    free_copies.up();
    // ... then wakeup waiting thread
}
```

counting resources: reserving books

suppose tracking copies of same library book

non-negative integer count = # how many books used?

up = give back book; down = take book

Copy 1
Copy 2
Copy 3

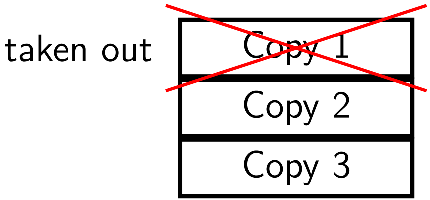
free copies 3

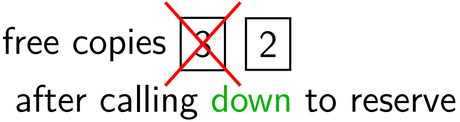
counting resources: reserving books

suppose tracking copies of same library book

non-negative integer count = # how many books used?

up = give back book; down = take book



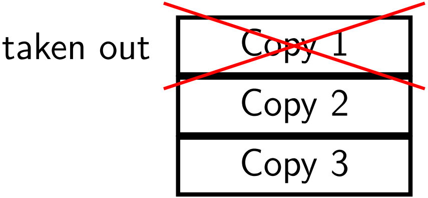


counting resources: reserving books

suppose tracking copies of same library book

non-negative integer count = # how many books used?

up = give back book; down = take book



free copies 2

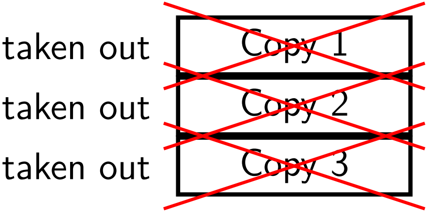
after calling down to reserve

counting resources: reserving books

suppose tracking copies of same library book

non-negative integer count = # how many books used?

up = give back book; down = take book

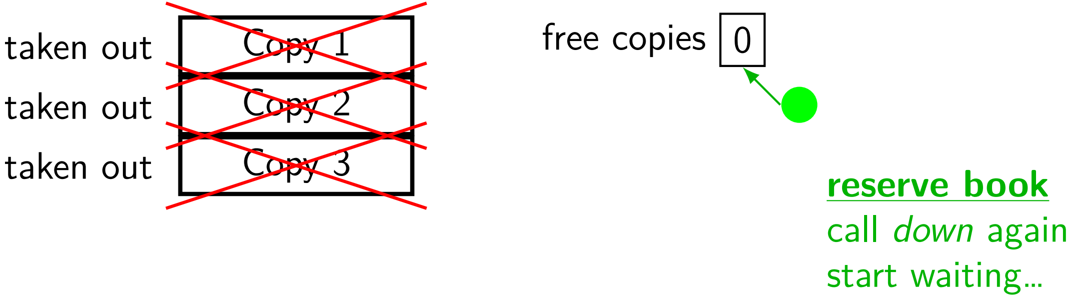


free copies 0

after calling down three times to reserve all copies

counting resources: reserving books

suppose tracking copies of same library book
non-negative integer count = # how many books used?
up = give back book; down = take book



counting resources: reserving books

suppose tracking copies of same library book

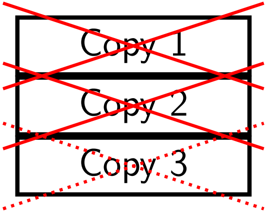
non-negative integer count = # how many books used?

up = give back book; down = take book

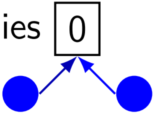
taken out

taken out

taken out



free copies



return book

call *up*

release waiter

reserve book

call *down*

~~waiting~~

done waiting

implementing mutexes with semaphores

```
struct Mutex {
    Semaphore s; /* with inital value 1 */
    /* value = 1 --> mutex if free */
    /* value = 0 --> mutex is busy */
}

MutexLock(Mutex *m) {
    m->s.down();
}

MutexUnlock(Mutex *m) {
    m->s.up();
}
```

implementing join with semaphores

```
struct Thread {
    ...
    Semaphore finish_semaphore; /* with initial value 0 */
    /* value = 0: either thread not finished OR already joined */
    /* value = 1: thread finished AND not joined */
};
thread_join(Thread *t) {
    t->finish_semaphore->down();
}

/* assume called when thread finishes */
thread_exit(Thread *t) {
    t->finish_semaphore->up();
    /* tricky part: deallocating struct Thread safely? */
}
```

POSIX semaphores

```
#include <semaphore.h>
...
sem_t my_semaphore;
int process_shared = /* 1 if sharing between processes */;
sem_init(&my_semaphore, process_shared, initial_value);
...
sem_wait(&my_semaphore);  /* down */
sem_post(&my_semaphore);  /* up */
...
sem_destroy(&my_semaphore);
```

semaphore exercise

```
int value;  sem_t empty, ready;  // with some initial values

void PutValue(int argument) {
    sem_wait(&empty);
    value = argument;
    sem_post(&ready);
}

int GetValue() {
    int result;
    _________________
    result = value;
    _________________
    return result;
}
```

What goes in the blanks?

A: `sem_post(&empty)` / `sem_wait(&ready)`

B: `sem_wait(&ready)` / `sem_post(&empty)`

C: `sem_post(&ready)` / `sem_wait(&empty)`

D: `sem_post(&ready)` / `sem_post(&empty)`

E: `sem_wait(&empty)` / `sem_post(&ready)`

F: something else

GetValue() waits for PutValue() to happen, retrieves value, then allows next PutValue().

PutValue() waits for prior GetValue(), places value, then allows next GetValue().

semaphore exercise [solution]

```
int value;
sem_t empty, ready;
void PutValue(int argument) {
    sem_wait(&empty);
    value = argument;
    sem_post(&ready);
}
int GetValue() {
    int result;
    sem_wait(&ready);
    result = value;
    sem_post(&empty);
    return result;
}
```

semaphore intuition

What do you need to wait for?

- critical section to be finished
- queue to be non-empty
- array to have space for new items

what can you count that will be 0 when you need to wait?

- # of threads that can start critical section now
- # of threads that can join another thread without waiting
- # of items in queue
- # of empty spaces in array

use up/down operations to maintain count

producer/consumer constraints

consumer waits for producer(s) if buffer is empty

producer waits for consumer(s) if buffer is full

any thread waits while a thread is manipulating the buffer

producer/consumer constraints

consumer waits for producer(s) if buffer is empty

producer waits for consumer(s) if buffer is full

any thread waits while a thread is manipulating the buffer

one semaphore per constraint:

```
sem_t full_slots;    // consumer waits if empty
sem_t empty_slots;   // producer waits if full
sem_t mutex;         // either waits if anyone changing buffer
FixedSizedQueue buffer;
```

producer/consumer pseudocode

```
sem_init(&full_slots, ..., 0 /* # buffer slots initially used */);
sem_init(&empty_slots, ..., BUFFER_CAPACITY);
sem_init(&mutex, ..., 1 /* # thread that can use buffer at once */);
buffer.set_size(BUFFER_CAPACITY);
...

Produce(item) {
    sem_wait(&empty_slots);  // wait until free slot, reserve it
    sem_wait(&mutex);
    buffer.enqueue(item);
    sem_post(&mutex);
    sem_post(&full_slots);  // tell consumers there is more data
}

Consume() {
    sem_wait(&full_slots);  // wait until queued item, reserve it
    sem_wait(&mutex);
    item = buffer.dequeue();
    sem_post(&mutex);
    sem_post(&empty_slots);  // let producer reuse item slot
    return item;
}
```

producer/consumer pseudocode

```
sem_init(&full_slots, ..., 0 /* # buffer slots initially used */);
sem_init(&empty_slots, ..., BUFFER_CAPACITY);
sem_init(&mutex, ..., 1 /* # thread that can use buffer at once */);
buffer.set_size(BUFFER_CAPACITY);
...

Produce(item) {
    sem_wait(&empty_slots);  // wait until free slot, reserve it
    sem_wait(&mutex);
    buffer.enqueue(item);
    sem_post(&mutex);
    sem_post(&full_slots);  // tell consumers there is more data
}

Consume() {
    sem_wait(&full_slots);  // wait until queued item, reserve it
    sem_wait(&mutex);
    item = buffer.dequeue();
    sem_post(&mutex);
    sem_post(&empty_slots);  // let producer reuse item slot
    return item;
}
```

producer/consumer pseudocode

```
sem_init(&full_slots, ..., 0 /* # buffer slots initially used */);
sem_init(&empty_slots, ..., BUFFER_CAPACITY);
sem_init(&mutex, ..., 1 /* # thread that can use buffer at once */);
buffer.set_size(BUFFER_CAPACITY);
...

Produce(item) {
    sem_wait(&empty_slots);  // wait until free slot, reserve it
    sem_wait(&mutex);
    buffer.enqueue(item);
    sem_post(&mutex);
    sem_post(&full_slots);  // tell consumers there is more data
}

Consume() {
    sem_wait(&full_slots);  // wait until queued item, reserve it
    sem_wait(&mutex);
    item = buffer.dequeue();
    sem_post(&mutex);
    sem_post(&empty_slots);  // let producer reuse item slot
    return item;
}
```

producer/consumer pseudocode

```
sem_init(&full_slots, ..., 0 /* # buffer slots initially used */);
sem_init(&empty_slots, ..., BUFFER_CAPACITY);
sem_init(&mutex, ..., 1 /* # thread that can use buffer at once */);
buffer.set_size(BUFFER_CAPACITY);
...

Produce(item) {
    sem_wait(&empty_slots);  // wait until free slot, reserve it
    sem_wait(&mutex);
    buffer.enqueue(item);
    sem_post(&mutex);
    sem_post(&full_slots);                             re data
}

Consume() {
    sem_wait(&full_slots);  // wait until queued item, reserve it
    sem_wait(&mutex);
    item = buffer.dequeue();
    sem_post(&mutex);
    sem_post(&empty_slots);  // let producer reuse item slot
    return item;
}
```

Can we do

```
sem_wait(&mutex);
sem_wait(&empty_slots);
```

instead?

producer/consumer pseudocode

```
sem_init(&full_slots, ..., 0 /* # buffer slots initially used */);
sem_init(&empty_slots, ..., BUFFER_CAPACITY);
sem_init(&mutex, ..., 1 /* # thread that can use buffer at once */);
buffer.set_size(BUFFER_CAPACITY);
...

Produce(item) {
    sem_wait(&empty_slots);  // wait until free slot, reserve it
    sem_wait(&mutex);
    buffer.enqueue(item);
    sem_post(&mutex);
    sem_post(&full_slots);                                   re data
}

Consume() {
    sem_wait(&full_slots);
    sem_wait(&mutex);
    item = buffer.dequeue()
    sem_post(&mutex);
    sem_post(&empty_slots);
    return item;
}
```

Can we do

```
sem_wait(&mutex);
sem_wait(&empty_slots);
```

instead?

No. Consumer waits on `sem_wait(&mutex)`

so can't `sem_post(&empty_slots)`

(result: producer waits forever

problem called *deadlock*)

producer/consumer: cannot reorder mutex/empty

```
ProducerReordered() {
  // BROKEN: WRONG ORDER
  sem_wait(&mutex);
  sem_wait(&empty_slots);

  ...

  sem_post(&mutex);
```

```
Consumer() {
  sem_wait(&full_slots);

  // can't finish until
  // Producer's sem_post(&mutex):
  sem_wait(&mutex);

  ...

  // so this is not reached
  sem_post(&full_slots);
```

producer/consumer pseudocode

```
sem_init(&full_slots, ..., 0 /* # buffer slots initially used */);
sem_init(&empty_slots, ..., BUFFER_CAPACITY);
sem_init(&mutex, ..., 1 /* # thread that can use buffer at once */);
buffer.set_size(BUFFER_CAPACITY);
...

Produce(item) {
    sem_wait(&empty_slots);  // wait until free slot, reserve it
    sem_wait(&mutex);
    buffer.enqueue(item);
    sem_post(&mutex);
    sem_post(&full_slots                        s more data
}

Consume() {
    sem_wait(&full_slots                        , reserve it
    sem_wait(&mutex);
    item = buffer.dequeu
    sem_post(&mutex);
    sem_post(&empty_slots);  // let producer reuse item slot
    return item;
}
```

Can we do

```
sem_post(&full_slots);
sem_post(&mutex);
```

instead?

Yes — post never waits

producer/consumer summary

producer: wait (down) empty_slots, post (up) full_slots

consumer: wait (down) full_slots, post (up) empty_slots

two producers or consumers?

- still works!

binary semaphores

binary semaphores — semaphores that are only zero or one

as powerful as normal semaphores

- exercise: simulate counting semaphores with binary semaphores (more than one) and an integer

counting semaphores with binary semaphores

via Hemmendinger, "Comments on 'A correect and unrestrictive implementation of general semaphores' " (1989); Barz, "Implementing semaphores by binary semaphores" (1983)

```
// assuming initialValue > 0
BinarySemaphore mutex(1);
int value = initialValue ;
BinarySemaphore gate(1 /* if initialValue >= 1 */);
    /* gate = # threads that can Down() now */
```

```
void Down() {
  gate.Down();
  // wait, if needed
  mutex.Down();
  value -= 1;
  if (value > 0) {
    gate.Up();
    // because next down should finish
    // now (but not marked to before)
  }
  mutex.Up();
}
```

```
void Up() {
  mutex.Down();
  value += 1;
  if (value == 1) {
    gate.Up();
    // because down should finish now
    // but could not before
  }
  mutex.Up();
}
```

gate intuition/pattern

pattern to allow one thread at a time:

```
sem_t gate; // 0 = closed; 1 = open
ReleasingThread() {
    ... // finish what the other thread is waiting for
    while (another thread is waiting and can go) {
        sem_post(&gate)  // allow EXACTLY ONE thread
        ... // other bookkeeping
    }
    ...
}
WaitingThread() {
    ... // indicate that we're waiting
    sem_wait(&gate) // wait for gate to be open
    ... // indicate that we're not waiting
}
```

Anderson-Dahlin and semaphores

Anderson/Dahlin complains about semaphores

> "Our view is that programming with locks and condition variables is superior to programming with semaphores."

argument 1: clearer to have separate constructs for

- waiting for condition to be come true, and
- allowing only one thread to manipulate a thing at a time

arugment 2: tricky to verify thread calls up exactly once for every down

- alternatives allow one to be sloppier (in a sense)

monitors with semaphores: locks

```
sem_t semaphore;  // initial value 1

Lock() {
    sem_wait(&semaphore);
}

Unlock() {
    sem_post(&semaphore);
}
```

monitors with semaphores: [broken] cvs

start with only wait/signal:

```
sem_t threads_to_wakeup;  // initially 0
Wait(Lock lock) {
    lock.Unlock();
    sem_wait(&threads_to_wakeup);
    lock.Lock();
}
Signal() {
    sem_post(&threads_to_wakeup);
}
```

monitors with semaphores: [broken] cvs

start with only wait/signal:

```
sem_t threads_to_wakeup;  // initially 0
Wait(Lock lock) {
    lock.Unlock();
    sem_wait(&threads_to_wakeup);
    lock.Lock();
}
Signal() {
    sem_post(&threads_to_wakeup);
}
```

problem: signal wakes up non-waiting threads (in the far future)

monitors with semaphores: cvs (better)

start with only wait/signal:

```
sem_t private_lock;  // initially 1
int num_waiters;
sem_t threads_to_wakeup;  // initially 0
Wait(Lock lock) {
  sem_wait(&private_lock);
  ++num_waiters;
  sem_post(&private_lock);
  lock.Unlock();
  sem_wait(&threads_to_wakeup);
  lock.Lock();
}
```

```
Signal() {
  sem_wait(&private_lock);
  if (num_waiters > 0) {
    sem_post(&threads_to_wakeup);
    --num_waiters;
  }
  sem_post(&private_lock);
}
```

monitors with semaphores: broadcast

now allows broadcast:

```
sem_t private_lock;  // initially 1
int num_waiters;
sem_t threads_to_wakeup;  // initially 0
Wait(Lock lock) {
  sem_wait(&private_lock);
  ++num_waiters;
  sem_post(&private_lock);
  lock.Unlock();
  sem_wait(&threads_to_wakeup);
  lock.Lock();
}
```

```
Broadcast() {
  sem_wait(&private_lock);
  while (num_waiters > 0) {
    sem_post(&threads_to_wakeup);
    --num_waiters;
  }
  sem_post(&private_lock);
}
```

building semaphore with monitors

```
pthread_mutex_t lock;
```

lock to protect shared state

building semaphore with monitors

```
pthread_mutex_t lock;
unsigned int count;
```

lock to protect shared state

- shared state: semaphore tracks a count

building semaphore with monitors

```
pthread_mutex_t lock;
unsigned int count;
/* condition, broadcast when becomes count > 0 */
pthread_cond_t count_is_positive_cv;
```

lock to protect shared state

- shared state: semaphore tracks a count

add cond var for each reason we wait

- semaphore: wait for count to become positive (for down)

building semaphore with monitors

```
pthread_mutex_t lock;
unsigned int count;
/* condition, broadcast when becomes count > 0 */
pthread_cond_t count_is_positive_cv;
void down() {
    pthread_mutex_lock(&lock);
    while (!(count > 0)) {
        pthread_cond_wait(
            &count_is_positive_cv,
            &lock);
    }
    count -= 1;
    pthread_mutex_unlock(&lock);
}
```

lock to protect shared state

- shared state: semaphore tracks a count

add cond var for each reason we wait

- semaphore: wait for count to become positive (for down)

wait using condvar; broadcast/signal when condition changes

building semaphore with monitors

```
pthread_mutex_t lock;
unsigned int count;
/* condition, broadcast when becomes count > 0 */
pthread_cond_t count_is_positive_cv;
void down() {
    pthread_mutex_lock(&lock);
    while (!(count > 0)) {
        pthread_cond_wait(
            &count_is_positive_cv,
            &lock);
    }
    count -= 1;
    pthread_mutex_unlock(&lock);
}
```

```
void up() {
    pthread_mutex_lock(&lock);
    count += 1;
    /* count must now be
       positive, and at most
       one thread can go per
       call to Up() */
    pthread_cond_signal(
        &count_is_positive_cv
    );
    pthread_mutex_unlock(&lock);
}
```

lock to protect shared state

- shared state: semaphore tracks a count

add cond var for each reason we wait

- semaphore: wait for count to become positive (for down)

wait using condvar; broadcast/signal when condition changes

reader/writer problem

some shared data

only one thread modifying (read+write) at a time

read-only access from multiple threads is safe

reader/writer problem

some shared data

only one thread modifying (read+write) at a time

read-only access from multiple threads is safe

could use lock — but doesn't allow multiple readers

reader/writer locks

abstraction: lock that distinguishes readers/writers

operations:

- read lock: wait until no writers
- read unlock: stop being registered as reader
- write lock: wait until no readers and no writers
- write unlock: stop being registered as writer

reader/writer locks

abstraction: lock that distinguishes readers/writers

operations:

- read lock: wait until no writers
- read unlock: stop being registered as reader
- write lock: wait until no readers and no writers
- write unlock: stop being registered as writer

pthread rwlocks

```
pthread_rwlock_t rwlock;
pthread_rwlock_init(&rwlock, NULL /* attributes */);
...
    pthread_rwlock_rdlock(&rwlock);
    ...  /* read shared data */
    pthread_rwlock_unlock(&rwlock);

    pthread_rwlock_wrlock(&rwlock);
    ...  /* read+write shared data */
    pthread_rwlock_unlock(&rwlock);

...
pthread_rwlock_destroy(&rwlock);
```

rwlocks with monitors (attempt 1)

```
mutex_t lock;
```

lock to protect shared state

rwlocks with monitors (attempt 1)

```
mutex_t lock;
unsigned int readers, writers;
```

state: number of active readers, writers

rwlocks with monitors (attempt 1)

```
mutex_t lock;
unsigned int readers, writers;
/* condition, signal when writers becomes 0 */
cond_t ok_to_read_cv;
/* condition, signal when readers + writers becomes 0 */
cond_t ok_to_write_cv;
```

conditions to wait for (no readers or writers, no writers)

rwlocks with monitors (attempt 1)

```
mutex_t lock;
unsigned int readers, writers;
/* condition, signal when writers becomes 0 */
cond_t ok_to_read_cv;
/* condition, signal when readers + writers becomes 0 */
cond_t ok_to_write_cv;
```

```
ReadLock() {
  mutex_lock(&lock);
  while (writers != 0) {
    cond_wait(&ok_to_read_cv, &lock);
  }
  ++readers;
  mutex_unlock(&lock);
}
ReadUnlock() {
  mutex_lock(&lock);
  --readers;
  if (readers == 0) {
    cond_signal(&ok_to_write_cv);
  }
  mutex_unlock(&lock);
}
```

```
WriteLock() {
  mutex_lock(&lock);
  while (readers + writers != 0) {
    cond_wait(&ok_to_write_cv);
  }
  ++writers;
  mutex_unlock(&lock);
}
WriteUnlock() {
  mutex_lock(&lock);
  --writers;
  cond_signal(&ok_to_write_cv);
  cond_broadcast(&ok_to_read_cv);
  mutex_unlock(&lock);
}
```

broadcast — wakeup all readers when no writers

rwlocks with monitors (attempt 1)

```
mutex_t lock;
unsigned int readers, writers;
/* condition, signal when writers becomes 0 */
cond_t ok_to_read_cv;
/* condition, signal when readers + writers becomes 0 */
cond_t ok_to_write_cv;
ReadLock() {
  mutex_lock(&lock);
  while (writers != 0) {
    cond_wait(&ok_to_read_cv, &lock);
  }
  ++readers;
  mutex_unlock(&lock);
}
ReadUnlock() {
  mutex_lock(&lock);
  --readers;
  if (readers == 0) {
    cond_signal(&ok_to_write_cv);
  }
  mutex_unlock(&lock);
}
```

```
WriteLock() {
  mutex_lock(&lock);
  while (readers + writers != 0) {
    cond_wait(&ok_to_write_cv);
  }
  ++writers;
  mutex_unlock(&lock);
}
WriteUnlock() {
  mutex_lock(&lock);
  --writers;
  cond_signal(&ok_to_write_cv);
  cond_broadcast(&ok_to_read_cv);
  mutex_unlock(&lock);
}
```

wakeup a single writer when no readers or writers

rwlocks with monitors (attempt 1)

```
mutex_t lock;
unsigned int readers, writers;
/* condition, signal when writers becomes 0 */
cond_t ok_to_read_cv;
/* condition, signal when readers + writers becomes 0 */
cond_t ok_to_write_cv;
ReadLock() {
  mutex_lock(&lock);
  while (writers != 0) {
    cond_wait(&ok_to_read_cv, &lock);
  }
  ++readers;
  mutex_unlock(&lock);
}
ReadUnlock() {
  mutex_lock(&lock);
  --readers;
  if (readers == 0) {
    cond_signal(&ok_to_write_cv);
  }
  mutex_unlock(&lock);
}
```

```
WriteLock() {
  mutex_lock(&lock);
  while (readers + writers != 0) {
    cond_wait(&ok_to_write_cv);
  }
  ++writers;
  mutex_unlock(&lock);
}
WriteUnlock() {
  mutex_lock(&lock);
  --writers;
  cond_signal(&ok_to_write_cv);
  cond_broadcast(&ok_to_read_cv);
  mutex_unlock(&lock);
}
```

problem: wakeup readers first or writer first?

this solution: wake them all up and they fight! inefficient!

reader/writer-priority

policy question: writers first or readers first?

- writers-first: no readers go when writer waiting
- readers-first: no writers go when reader waiting

previous implementation: whatever randomly happens

- writers signalled first, maybe gets lock first?
- ...but non-determinstic in pthreads

can make explicit decision

reader/writer-priority

policy question: writers first or readers first?

- writers-first: no readers go when writer waiting
- readers-first: no writers go when reader waiting

previous implementation: whatever randomly happens

- writers signalled first, maybe gets lock first?
- ...but non-determinstic in pthreads

can make explicit decision

key method: track number of waiting readers/writers

writer-priority (1)

```
mutex_t lock; cond_t ok_to_read_cv; cond_t ok_to_write_cv;
int readers = 0, writers = 0;
int waiting_writers = 0;
ReadLock() {
  mutex_lock(&lock);
  while (writers != 0
         || waiting_writers != 0) {
    cond_wait(&ok_to_read_cv, &lock);
  }
  ++readers;
  mutex_unlock(&lock);
}

ReadUnlock() {
  mutex_lock(&lock);
  --readers;
  if (readers == 0) {
    cond_signal(&ok_to_write_cv);
  }
  mutex_unlock(&lock);
}
```

```
WriteLock() {
  mutex_lock(&lock);
  ++waiting_writers;
  while (readers + writers != 0) {
    cond_wait(&ok_to_write_cv, &lock);
  }
  --waiting_writers;
  ++writers;
  mutex_unlock(&lock);
}

WriteUnlock() {
  mutex_lock(&lock);
  --writers;
  if (waiting_writers != 0) {
    cond_signal(&ok_to_write_cv);
  } else {
    cond_broadcast(&ok_to_read_cv);
  }
  mutex_unlock(&lock);
}
```

writer-priority (1)

```
mutex_t lock; cond_t ok_to_read_cv; cond_t ok_to_write_cv;
int readers = 0, writers = 0;
int waiting_writers = 0;
ReadLock() {
  mutex_lock(&lock);
  while (writers != 0
         || waiting_writers != 0) {
    cond_wait(&ok_to_read_cv, &lock);
  }
  ++readers;
  mutex_unlock(&lock);
}

ReadUnlock() {
  mutex_lock(&lock);
  --readers;
  if (readers == 0) {
    cond_signal(&ok_to_write_cv);
  }
  mutex_unlock(&lock);
}
```

```
WriteLock() {
  mutex_lock(&lock);
  ++waiting_writers;
  while (readers + writers != 0) {
    cond_wait(&ok_to_write_cv, &lock);
  }
  --waiting_writers;
  ++writers;
  mutex_unlock(&lock);
}

WriteUnlock() {
  mutex_lock(&lock);
  --writers;
  if (waiting_writers != 0) {
    cond_signal(&ok_to_write_cv);
  } else {
    cond_broadcast(&ok_to_read_cv);
  }
  mutex_unlock(&lock);
}
```

writer-priority (1)

```
mutex_t lock; cond_t ok_to_read_cv; cond_t ok_to_write_cv;
int readers = 0, writers = 0;
int waiting_writers = 0;
ReadLock() {
  mutex_lock(&lock);
  while (writers != 0
        || waiting_writers != 0) {
    cond_wait(&ok_to_read_cv, &lock);
  }
  ++readers;
  mutex_unlock(&lock);
}

ReadUnlock() {
  mutex_lock(&lock);
  --readers;
  if (readers == 0) {
    cond_signal(&ok_to_write_cv);
  }
  mutex_unlock(&lock);
}
```

```
WriteLock() {
  mutex_lock(&lock);
  ++waiting_writers;
  while (readers + writers != 0) {
    cond_wait(&ok_to_write_cv, &lock);
  }
  --waiting_writers;
  ++writers;
  mutex_unlock(&lock);
}

WriteUnlock() {
  mutex_lock(&lock);
  --writers;
  if (waiting_writers != 0) {
    cond_signal(&ok_to_write_cv);
  } else {
    cond_broadcast(&ok_to_read_cv);
  }
  mutex_unlock(&lock);
}
```

simulation of reader/write lock

writer-priority version

W = writers, R = readers, WW = waiting_writers

reader 1	reader 2	writer 1	reader 3	W	R	WW
				0	0	0

simulation of reader/write lock

writer-priority version

W = writers, R = readers, WW = waiting_writers

reader 1	reader 2	writer 1	reader 3	W	R	WW
				0	0	0

simulation of reader/write lock

writer-priority version

W = writers, R = readers, WW = waiting_writers

reader 1	reader 2	writer 1	reader 3	W	R	WW
				0	0	0
ReadLock				0	1	0

```
mutex_lock(&lock);
while (writers != 0 || waiting_writers != 0) {
  cond_wait(&ok_to_read_cv, &lock);
}
++readers;
mutex_unlock(&lock);
```

simulation of reader/write lock

writer-priority version

W = writers, R = readers, WW = waiting_writers

reader 1	reader 2	writer 1	reader 3	W	R	WW
				0	0	0
ReadLock				0	1	0
(reading)	ReadLock			0	2	0

simulation of reader/write lock

writer-priority version

W = writers, R = readers, WW = waiting_writers

reader 1	reader 2	writer 1	reader 3	W	R	WW
				0	0	0
ReadLock				0	1	0
(reading)	ReadLock			0	2	0
(reading)	(reading)	WriteLock wait		0	2	1

```
mutex_lock(&lock);
++waiting_writers;
while (readers + writers != 0) {
  cond_wait(&ok_to_write_cv, &lock);
}
```

simulation of reader/write lock

writer-priority version

W = writers, R = readers, WW = waiting_writers

reader 1	reader 2	writer 1	reader 3	W	R	WW
				0	0	0
ReadLock				0	1	0
(reading)	ReadLock			0	2	0
(reading)	(reading)	WriteLock wait		0	2	1
(reading)	(reading)	WriteLock wait	ReadLock wait	0	2	1

simulation of reader/write lock

writer-priority version

W = writers, R = readers, WW = waiting_writers

reader 1	reader 2	writer 1	reader 3	W	R	WW
				0	0	0
ReadLock				0	1	0
(reading)	ReadLock			0	2	0
(reading)	(reading)	WriteLock wait		0	2	1
(reading)	(read	wait	ReadLock wait	0	2	1
ReadUnlock	(r	wait	ReadLock wait	0	1	1

```
mutex_lock(&lock);
--readers;
if (readers == 0)
  ...
```

simulation of reader/write lock

writer-priority version

W = writers, R = readers, WW = waiting_writers

reader 1	reader 2	writer 1	reader 3	W	R	WW
				0	0	0
ReadLock				0	1	0
(reading)	ReadLock			0	2	0
(reading)	(reading)	WriteLock wait		0	2	1
(reading)	(reading)	WriteLock wait	ReadLock wait	0	2	1
ReadUnlock	(reading)	Write			1	1
	ReadUnlock	Wri			0	1

```
mutex_lock(&lock);
--readers;
if (readers == 0)
  cond_signal(&ok_to_write_cv)
mutex_unlock(&lock);
```

simulation of reader/write lock

writer-priority version

W = writers, R = readers, WW = waiting_writers

reader 1	reader 2	writer 1	reader 3	W	R	WW
				0	0	0
ReadLock				0	1	0
(reading)	Read			0	2	0
(reading)	(rea			0	2	1
(reading)	(rea		it	0	2	1
ReadUnlock	(reading)	WriteLock wait	ReadLock wait	0	1	1
	ReadUnlock	WriteLock wait	ReadLock wait	0	0	1
		WriteLock	ReadLock wait	1	0	0

```
while (readers + writers != 0) {
  cond_wait(&ok_to_write_cv, &lock);
}
--waiting_writers; ++writers;
mutex_unlock(&lock);
```

simulation of reader/write lock

writer-priority version

W = writers, R = readers, WW = waiting_writers

reader 1	reader 2	writer 1	reader 3	W	R	WW
				0	0	0
ReadLock				0	1	0
(reading)	ReadLock			0	2	0
(reading)	(reading)	WriteLock wait		0	2	1
(reading)	(reading)	WriteLock wait	ReadLock wait	0	2	1
ReadUnlock	(reading)	WriteLock wait	ReadLock wait	0	1	1
	ReadUnlock	WriteLock wait	ReadLock wait	0	0	1
		WriteLock	ReadLock wait	1	0	0
		(read+writing)	ReadLock wait	1	0	0

simulation of reader/write lock

writer-priority version

W = writers, R = readers, WW = waiting_writers

reader 1	reader 2	writer 1	reader 3	W	R	WW
				0	0	0
ReadLock				0	1	0
(reading)	ReadLo			0	2	0
(reading)	(readi			0	2	1
(reading)	(readi		wait	0	2	1
ReadUnlock	(readi		wait	0	1	1
	ReadUr		wait	0	0	1
		WriteLo k	ReadLock wait	1	0	0
		(read+w iting)	ReadLock wait	1	0	0
		WriteUnlock	ReadLock wait	0	0	0

```
mutex_lock(&lock);
if (waiting_writers != 0) {
  cond_signal(&ok_to_write_cv);
} else {
  cond_broadcast(&ok_to_read_cv);
}
```

simulation of reader/write lock

writer-priority version

W = writers, R = readers, WW = waiting_writers

reader 1	reader 2	writer 1	reader 3	W	R	WW
				0	0	0
ReadLock				0	1	0
(reading)	ReadLock			0	2	0
(reading)	(reading)					
(reading)	(reading)					
ReadUnlock	(reading)					
	ReadUnlock					
		WriteLock	ReadLock wait	1	0	0
		(read+writing)	ReadLock wait	1	0	0
		WriteUnlock	ReadLock wait	0	0	0
			ReadLock	0	1	0

```
while (writers != 0 && waiting_writers != 0) {
  cond_wait(&ok_to_read_cv, &lock);
}
++readers;
mutex_unlock(&lock);
```

simulation of reader/write lock

writer-priority version

W = writers, R = readers, WW = waiting_writers

reader 1	reader 2	writer 1	reader 3	W	R	WW
				0	0	0
ReadLock				0	1	0
(reading)	ReadLock			0	2	0
(reading)	(reading)	WriteLock wait		0	2	1
(reading)	(reading)	WriteLock wait	ReadLock wait	0	2	1
ReadUnlock	(reading)	WriteLock wait	ReadLock wait	0	1	1
	ReadUnlock	WriteLock wait	ReadLock wait	0	0	1
		WriteLock	ReadLock wait	1	0	0
		(read+writing)	ReadLock wait	1	0	0
		WriteUnlock	ReadLock wait	0	0	0
			ReadLock	0	1	0

reader-priority (1)

```
...
int waiting_readers = 0;
ReadLock() {
  mutex_lock(&lock);
  ++waiting_readers;
  while (writers != 0) {
    cond_wait(&ok_to_read_cv, &lock);
  }
  --waiting_readers;
  ++readers;
  mutex_unlock(&lock);
}

ReadUnlock() {
  ...
  if (waiting_readers == 0) {
    cond_signal(&ok_to_write_cv);
  }
}
```

```
WriteLock() {
  mutex_lock(&lock);
  while (waiting_readers +
            readers + writers != 0) {
    cond_wait(&ok_to_write_cv);
  }
  ++writers;
  mutex_unlock(&lock);
}
WriteUnlock() {
  mutex_lock(&lock);
  --writers;
  if (readers == 0 && waiting_readers == 0) {
    cond_signal(&ok_to_write_cv);
  } else {
    cond_broadcast(&ok_to_read_cv);
  }
  mutex_unlock(&lock);
}
```

reader-priority (1)

```
...
int waiting_readers = 0;
ReadLock() {
  mutex_lock(&lock);
  ++waiting_readers;
  while (writers != 0) {
    cond_wait(&ok_to_read_cv, &lock);
  }
  --waiting_readers;
  ++readers;
  mutex_unlock(&lock);
}

ReadUnlock() {
  ...
  if (waiting_readers == 0) {
    cond_signal(&ok_to_write_cv);
  }
}
```

```
WriteLock() {
  mutex_lock(&lock);
  while (waiting_readers +
         readers + writers != 0) {
    cond_wait(&ok_to_write_cv);
  }
  ++writers;
  mutex_unlock(&lock);
}
WriteUnlock() {
  mutex_lock(&lock);
  --writers;
  if (readers == 0 && waiting_readers == 0) {
    cond_signal(&ok_to_write_cv);
  } else {
    cond_broadcast(&ok_to_read_cv);
  }
  mutex_unlock(&lock);
}
```

rwlock exercise

suppose we want something in-between reader and writer priority:

reader-priority except if writers wait more than 1 second

exercise: what do we change?

```
...
int waiting_readers = 0;
ReadLock() {
  mutex_lock(&lock);
  ++waiting_readers;
  while (writers != 0) {
    cond_wait(&ok_to_read_cv, &lock);
  }
  --waiting_readers;
  ++readers;
  mutex_unlock(&lock);
}

ReadUnlock() {
  mutex_lock(&lock);
  --readers;
  if (waiting_readers == 0) {
    cond_signal(&ok_to_write_cv);
  }
  mutex_unlock(&lock);
```

```
WriteLock() {
  mutex_lock(&lock);
  while (waiting_readers + readers + writers != 0) {
    cond_wait(&ok_to_write_cv);
  }
  ++writers;
  mutex_unlock(&lock);
}
WriteUnlock() {
  mutex_lock(&lock);
  --writers;
  if (waiting_readers == 0) {
    cond_signal(&ok_to_write_cv);
  } else {
    cond_broadcast(&ok_to_read_cv);
  }
  mutex_unlock(&lock);
}
```

backup slides

monitor pattern

```
pthread_mutex_lock(&lock);
while (!condition A) {
    pthread_cond_wait(&condvar_for_A, &lock);
}
... /* manipulate shared data, changing other conditions */
if (set condition B) {
    pthread_cond_broadcast(&condvar_for_B);
    /* or signal, if only one thread cares */
}
if (set condition C) {
    pthread_cond_broadcast(&condvar_for_C);
    /* or signal, if only one thread cares */
}
...
pthread_mutex_unlock(&lock)
```

monitors rules of thumb

never touch shared data without holding the lock

keep lock held for entire operation:

> verifying condition (e.g. buffer not full) *up to and including* manipulating data (e.g. adding to buffer)

create condvar for every kind of scenario waited for

always write loop calling cond_wait to wait for condition X

broadcast/signal condition variable every time you change X

monitors rules of thumb

never touch shared data without holding the lock

keep lock held for entire operation:

verifying condition (e.g. buffer not full) *up to and including*
manipulating data (e.g. adding to buffer)

create condvar for every kind of scenario waited for

always write loop calling cond_wait to wait for condition X

broadcast/signal condition variable every time you change X

correct but slow to...

broadcast when just signal would work
broadcast or signal when nothing changed
use one condvar for multiple conditions

mutex/cond var init/destroy

```
pthread_mutex_t mutex;
pthread_cond_t cv;
pthread_mutex_init(&mutex, NULL);
pthread_cond_init(&cv, NULL);
// --OR--
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cv = PTHREAD_COND_INITIALIZER;

// and when done:
...
pthread_cond_destroy(&cv);
pthread_mutex_destroy(&mutex);
```

semaphores/CV

```
int num_waiting = 0;
bool finished = false;
sem_t mutex; // initially 1
sem_t gate;  // initially 0
void WaitForFinished() {
    sem_wait(&mutex);
    if (finished) {
        sem_post(&mutex);
    } else {
        num_waiting += 1;
        sem_post(&mutex);
        sem_wait(&gate);
    }
}

void Finish() {
    sem_wait(&mutex);
    finished = true;
    while (num_waiting > 0) {
        num_waiting -= 1;
        sem_post(&gate);
    }
}
```

```
bool finished = false;
pthread_mutex_t mutex;
pthread_cond_t cv;

void WaitForFinished() {
    pthread_mutex_lock(&mutex);
    while (!finished) {
        pthread_cond_wait(&cv, &mutex);
    }
    pthread_mutex_unlock(&mutex);
}

void Finish() {
    pthread_mutex_lock(&mutex);
    finished = true;
    pthread_cond_broadcast(&cv);
    pthread_mutex_unlock(&mutex);
}
```

semaphores/CV

```
int num_waiting = 0;
bool finished = false;
sem_t mutex; // initially 1
sem_t gate;  // initially 0
void WaitForFinished() {
    sem_wait(&mutex);
    if (finished) {
        sem_post(&mutex);
    } else {
        num_waiting += 1;
        sem_post(&mutex);
        sem_wait(&gate);
    }
}

void Finish() {
    sem_wait(&mutex);
    finished = true;
    while (num_waiting > 0) {
        num_waiting -= 1;
        sem_post(&gate);
    }
}
```

```
bool finished = false;
pthread_mutex_t mutex;
pthread_cond_t cv;


void WaitForFinished() {
    pthread_mutex_lock(&mutex);
    while (!finished) {
        pthread_cond_wait(&cv, &mutex);
    }
    pthread_mutex_unlock(&mutex);
}


void Finish() {
    pthread_mutex_lock(&mutex);
    finished = true;
    pthread_cond_broadcast(&cv);
    pthread_mutex_unlock(&mutex);
}
```

monitors with semaphores: chosen order

if we want to make sure threads woken up in order

```
ThreadSafeQueue<sem_t> waiters;
Wait(Lock lock) {
  sem_t private_semaphore;
  ... /* init semaphore
        with count 0 */
  waiters.Enqueue(&semaphore);
  lock.Unlock();
  sem_post(private_semaphore);
  lock.Lock();
}
```

```
Signal() {
  sem_t *next = waiters.DequeueOrNull();
  if (next != NULL) {
    sem_post(next);
  }
}
```

monitors with semaphores: chosen order

if we want to make sure threads woken up in order

```
ThreadSafeQueue<sem_t> waiters;
Wait(Lock lock) {
  sem_t private_semaphore;
  ... /* init semaphore
        with count 0 */
  waiters.Enqueue(&semaphore);
  lock.Unlock();
  sem_post(private_semaphore);
  lock.Lock();
}
```

```
Signal() {
  sem_t *next = waiters.DequeueOrNull();
  if (next != NULL) {
    sem_post(next);
  }
}
```

(but now implement queue with semaphores...)

rwlock exercise (1)

suppose there are multiple waiting writers

which one gets waken up first?

- whichever gets signal'd or gets lock first

could instead keep in order they started waiting

exercise: what extra information should we track?

- hint: we might need an array

```
mutex_t lock; cond_t ok_to_read_cv, ok_to_write_cv;
int readers, writers, waiting_writers;
```

rwlock exercise solution?

list of waiting writes?

```
struct WaitingWriter {
    cond_t cv;
    bool ready;
};
Queue<WaitingWriter*> waiting_writers;

WriteLock(...) {
  ...
  if (need to wait) {
    WaitingWriter self;
    self.ready = false;
    ...
    while(!self.ready) {
        pthread_cond_wait(&self.cv, &lock);
    }
  }
  ...
}
```

rwlock exercise solution?

dedicated writing thread with queue

(DoWrite∼Produce; WritingThread∼Consume)

```
ThreadSafeQueue<WritingTask*> waiting_writes;
WritingThread() {
    while (true) {
        WritingTask* task = waiting_writer.Dequeue();
        WriteLock();
        DoWriteTask(task);
        task.done = true;
        cond_broadcast(&task.cv);
    }
}
DoWrite(task) {
    // instead of WriteLock(); DoWriteTask(...); WriteUnlock()
    WritingTask task = ...;
    waiting_writes.Enqueue(&task);
    while (!task.done) { cond_wait(&task.cv); }
}
```

building semaphore with monitors (version B)

```
pthread_mutex_t lock;
unsigned int count;
/* condition, broadcast when becomes count > 0 */
pthread_cond_t count_is_positive_cv;
void down() {
    pthread_mutex_lock(&lock);
    while (!(count > 0)) {
        pthread_cond_wait(
            &count_is_positive_cv,
            &lock);
    }
    count -= 1;
    pthread_mutex_unlock(&lock);
}
```

```
void up() {
    pthread_mutex_lock(&lock);
    count += 1;
    /* condition *just* became true */
    if (count == 1) {
        pthread_cond_broadcast(
            &count_is_positive_cv
        );
    }
    pthread_mutex_unlock(&lock);
}
```

before: signal every time

can check if condition just became true instead?

building semaphore with monitors (version B)

```
pthread_mutex_t lock;
unsigned int count;
/* condition, broadcast when becomes count > 0 */
pthread_cond_t count_is_positive_cv;
void down() {
    pthread_mutex_lock(&lock);
    while (!(count > 0)) {
        pthread_cond_wait(
            &count_is_positive_cv,
            &lock);
    }
    count -= 1;
    pthread_mutex_unlock(&lock);
}
```

```
void up() {
    pthread_mutex_lock(&lock);
    count += 1;
    /* condition *just* became true */
    if (count == 1) {
        pthread_cond_broadcast(
            &count_is_positive_cv
        );
    }
    pthread_mutex_unlock(&lock);
}
```

before: signal every time

can check if condition just became true instead?

but do we really need to broadcast?

exercise: why broadcast?

```
pthread_mutex_t lock;
unsigned int count;
/* condition, broadcast when becomes count > 0 */
pthread_cond_t count_is_positive_cv;
void down() {
    pthread_mutex_lock(&lock);
    while (!(count > 0)) {
        pthread_cond_wait(
            &count_is_positive_cv,
            &lock);
    }
    count -= 1;
    pthread_mutex_unlock(&lock);
}
```

```
void up() {
    pthread_mutex_lock(&lock);
    count += 1;
    if (count == 1) { /* became > 0 */
        pthread_cond_broadcast(
            &count_is_positive_cv
        );
    }
    pthread_mutex_unlock(&lock);
}
```

exercise: why can't this be `pthread_cond_signal`?

hint: think of two threads calling down + two calling up?

brute force: only so many orders they can get the lock in

broadcast problem

Thread 1	Thread 2	Thread 3	Thread 4
Down()			
lock			
count == 0? yes			
unlock/wait			
	Down()		
	lock		
	count == 0? yes		
	unlock/wait		
		Up()	
		lock	
		count += 1 (now 1)	Up()
stop waiting on CV		signal	wait for lock
wait for lock		unlock	wait for lock
wait for lock			lock
wait for lock			count += 1 (now 2)
wait for lock			count != 1: don't signal
lock			unlock
count == 0? no			
count -= 1 (becomes 1)			
unlock			
	still waiting???		

broadcast problem

Thread 1	Thread 2	Thread 3	Thread 4
Down()			
lock			
count == 0? yes			
unlock/wait			
	Down()		
	lock		
	count == 0? yes		
	unlock/wait		
		Up()	
		lock	
		count += 1 (now 1)	Up()
stop waiting on CV		signal	wait for lock
wait for lock		unlock	wait for lock
wait for lock			lock
wait for lock			count += 1 (now 2)
wait for lock			count != 1: don't signal
lock			unlock
count == 0? no			
count -= 1 (becomes 1)			
unlock			
	still waiting???		

broadcast problem

Thread 1	Thread 2	Thread 3	Thread 4
Down()			
lock			
count == 0? yes			
unlock/wait			
	Down()		
	lock		
	count == 0? yes		
	unlock/wait		
		Up()	
		lock	
		count += 1 (now 1)	Up()
stop waiting on CV		signal	wait for lock
wait for lock		unlock	wait for lock
wait for lock			lock
wait for lock			count += 1 (now 2)
wait for lock			count != 1: don't signal
lock			unlock
count == 0? no			
count -= 1 (becomes 1)			
unlock			
	still waiting???		

Mesa-style monitors
signalling doesn't
“hand off” lock